The Link Foundation Fellowship in Advanced Simulation and Training

Software Tools for Solid Freeform Fabrication

by Robert Franceschini

Department of Computer Science
University of Central Florida

September 1998
The Link Foundation Fellowship in Advanced Simulation and Training

Software Tools for Solid Freeform Fabrication

by Robert Franceschini

Department of Computer Science
University of Central Florida

September 1998
The Link Foundation Fellowship in Advanced Simulation and Training

A grant is awarded to foster advanced level study in simulation and training research: to enhance and expand the theoretical and practical knowledge of how to train the operators and users of complex systems and how to simulate the real-world environments in which they function; and to disseminate the results of that research through lectures, seminars and publications. This publication is the final report of the fellow's research.

**LINK AWARDEES**

1991-92 Joe Dumas / University of Central Florida
1992-93 Julia Carrington / University of Central Florida
1992-93 Ginger Watson-Papelis / University of Iowa
1993-94 Michael Bajura / North Carolina University
1993-94 Eric Foxlin / Massachusetts Institute of Technology
1993-94 Gary R. Geroge / Binghamton University
1993-94 Matthew P. Reed / University of Michigan
1993-94 Benjamin A. Watson / Georgia Institute of Technology
1993-94 Ronald J. Guckenberger / University of Central Florida
1993-94 Henry A. Nelson / University of Central Florida
1994-95 Kurt M. Joseph / Kansas State University
1994-95 Mark R. Mine / University of North Carolina
1994-95 James R. Otto / University of Kentucky
1994-95 Jim Chen / University of Central Florida
1995-96 Ellen Bass / Georgia Institute of Technology
1995-96 Jonathan Cohen / University of North Carolina
1995-96 Doreen Comerford / Kansas State University
1995-96 Curtis Lisle / University of Central Florida
1995-96 Hans Weber / University of North Carolina
1995-96 Donna Wilt / Florida Institute of Technology
1996-97 Robert A. Glaser / University of Massachusetts
1996-97 Leslie L. Heimenz / Ohio State University
1996-97 Mark C. Kilby / University of Central Florida
1996-97 William R. Mark / University of North Carolina
1996-97 Ellen Scher Zagier / University of North Carolina
1996-97 Douglas A. Peterson / University of South Dakota
1997-98 Robert Franceschini / University of Central Florida
1997-98 Barry Kelly / Binghamton University
1997-98 Darren Law / University of Central Florida
1997-98 Mark Livingston / University of North Carolina
1997-98 Glenn Martin / University of Central Florida
1997-98 Carol Chensey / University of Florida

The Institute for Simulation and Training and the University of Central Florida
Administer this program on behalf of the Link Foundation.

Dr. A. Louis Medin, Executive Director
Institute for Simulation and Training
3280 Progress Drive
Orlando, Florida 32826-0544
SOFTWARE TOOLS FOR SOLID FREEFORM FABRICATION

Robert Franceschini, Lee Napravnik, and Amar Mukherjee
University of Central Florida
ABSTRACT

A structured design methodology played a key role in the VLSI (Very Large Scale Integration) revolution. In this methodology, the design is abstracted at different levels (viz. system, function, logic, circuit, and layout) taking into account concerns relevant to a particular level of abstraction. This allows development of CAD tools that can describe, synthesize, and simulate the design at high levels of abstraction independent of the details of the lower levels. In recent years, many researchers have observed that the lessons learned from the “VLSI experience” could be profitably applied to new classes of manufacturing processes called SFF (Solid Freeform Fabrication).

The advent of SFF has opened up new possibilities for rapid prototyping and agile distributed manufacturing. This thesis is concerned with developing tools capable of aiding the rapid prototyping process. We have developed an algorithm for the slicing of 3D polyhedral solid objects into layers capable of being passed to process planners for model fabrication. In support of the slicer algorithm, two new process planner algorithms have also been designed and implemented, one for LADRP (Laser Aided Direct Rapid Prototyping), and another for SLS (Selective Laser Sintering). These two new algorithms are able to accept model layers from the slicer, and convert the geometric information into command languages that can be understood by the laser controllers in each of the fabrication processes. The software tools have been linked to the fabrication system at the LAMMP lab at CREOL, and parts have been produced using the tools.
# TABLE OF CONTENTS

**LIST OF FIGURES** .................................................................................................................. 5  
**LIST OF SYMBOLS, ABBREVIATIONS, OR NOMENCLATURE** ............................................. 6  
**1 STRUCTURED DESIGN METHODOLOGY FOR SFF** ............................................................ 7  
**2 CONTRIBUTING TECHNOLOGIES** .................................................................................... 9  
  2.1 Laser Aided Direct Rapid Prototyping .............................................................................. 9  
  2.2 Selective Laser Sintering ................................................................................................. 13  
  2.3 Geometric Modeling ....................................................................................................... 17  
    2.3.1 Wireframe Models ........................................................................................................ 17  
    2.3.2 Surface Models .......................................................................................................... 18  
    2.3.3 Solid Models ............................................................................................................... 20  
    2.3.4 Constructive Solid Geometry ...................................................................................... 20  
    2.3.5 Boundary Representation ........................................................................................... 21  
    2.3.6 Associated Issues ........................................................................................................ 23  
    2.3.7 Winged Edge Representations .................................................................................... 24  
  2.4 Unigrafix .......................................................................................................................... 25  
    2.4.1 Unigrafix Data Structures and Syntax ........................................................................ 26  
  2.5 Design Hierarchy ............................................................................................................. 29  
**3 The Slicer** ............................................................................................................................ 31  
  3.1 General Algorithm Description ....................................................................................... 31  
  3.2 Slicer Algorithm ............................................................................................................. 32  
  3.3 Slicer Implementation ...................................................................................................... 33  
  3.4 Slicer Algorithm Analysis ............................................................................................... 37  
    3.4.1 Algorithm Analysis ...................................................................................................... 37  
    3.4.2 Implementation Analysis .......................................................................................... 37  
  3.5 Floating Point Error ......................................................................................................... 37  
**4 Process Planning** ................................................................................................................ 39  
  4.1 Laser Aided Direct Rapid Prototyping (LADRP) ............................................................. 39  
    4.1.1 LADRP Planner Algorithm ....................................................................................... 39  
    4.1.2 LADRP Planner Implementation ............................................................................... 40  
    4.1.3 LADRP Planner Algorithm Analysis ....................................................................... 42  
    4.1.4 LADRP Planner Implementation Analysis ............................................................... 42  
  4.2 Selective Laser Sintering (SLS) ....................................................................................... 43  
    4.2.1 SLS Process Planner Algorithm Specification ....................................................... 43  
    4.2.2 SLS Process Planner Algorithm Implementation .................................................... 44  
    4.2.3 SLS Specification Analysis ....................................................................................... 50  
    4.2.4 SLS Implementation Analysis ................................................................................... 51  
**5 Results** ................................................................................................................................. 52  
**6 Conclusion** .......................................................................................................................... 59  
**7 Future Work** ........................................................................................................................ 60
7.1 Unigrafix GUI................................................................. 60
7.2 BREP to CSG Converter .................................................. 61
7.3 Curved Surfaces................................................................. 61
7.4 Adaptive Slicing................................................................. 61
Bibliography ......................................................................... 62
LIST OF FIGURES

FIGURE 1: SAMPLE LADR setup ................................................................. 10
FIGURE 2: CUBE WITH EDGES LENGTH 2.0 ........................................... 11
FIGURE 3: SAMPLE LADR SLICE FILE .................................................. 13
FIGURE 4: SAMPLE SLS SETUP .............................................................. 14
FIGURE 5: CUBIC OBJECT WITH CUBIC HOLE .................................... 15
FIGURE 6: AMBIGUOUS WIREFRAME OBJECT ...................................... 18
FIGURE 7: SURFACE MODEL REPRESENTATION OF A PYRAMID ........... 19
FIGURE 8: REGULARIZED BOOLEAN OPERATIONS .................................. 20
FIGURE 9: CSG TREE OF AN OBJECT ................................................... 21
FIGURE 10: BREP REPRESENTATION OF AN OBJECT ............................ 22
FIGURE 11: WINGED EDGE DATA STRUCTURE ..................................... 25
FIGURE 12: SAMPLE UNIGRAFIX FILE, CUBE WITH HOLE ................. 27
FIGURE 13: DESIGN HIERARCHY ............................................................ 30
FIGURE 14: HEXAGONAL OBJECT WITH SWEEP PLANES ...................... 35
FIGURE 15: SLS HIT CASE ONE ............................................................ 46
FIGURE 16: SLS HIT CASE TWO ............................................................ 47
FIGURE 17: SLS HIT CASE THREE ........................................................ 48
FIGURE 18: CW/CCW PICTORIAL PROOF ............................................ 49
FIGURE 19: SAMPLE MODEL #1 ............................................................. 53
FIGURE 20: SLICES OF MODEL #1 ......................................................... 54
FIGURE 21: SAMPLE MODEL #2 ............................................................. 55
FIGURE 22: SLICES OF MODEL #2 ......................................................... 56
FIGURE 23: SAMPLE MODEL #3 ............................................................. 57
FIGURE 24: SLICES OF MODEL #3 ......................................................... 58
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREP</td>
<td>Boundary Representation</td>
</tr>
<tr>
<td>CCW</td>
<td>Counter Clockwise</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complimentary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CREOL</td>
<td>Center for Research and Education in Optics and Lasers</td>
</tr>
<tr>
<td>CSG</td>
<td>Constructive Solid Geometry</td>
</tr>
<tr>
<td>CW</td>
<td>Clockwise</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronic Engineers</td>
</tr>
<tr>
<td>LADRP</td>
<td>Laser Aided Direct Rapid Prototyping</td>
</tr>
<tr>
<td>L-SIF</td>
<td>Layered-Solid Interchange Format</td>
</tr>
<tr>
<td>NSF</td>
<td>National Science Foundation</td>
</tr>
<tr>
<td>NURBs</td>
<td>Non-uniform Rational B-spline's</td>
</tr>
<tr>
<td>SFF</td>
<td>Solid Free-Form Fabrication</td>
</tr>
<tr>
<td>SGI</td>
<td>Silicon Graphics Incorporated</td>
</tr>
<tr>
<td>SLS</td>
<td>Selective Laser Sintering</td>
</tr>
<tr>
<td>SIF</td>
<td>Solid Interchange Format</td>
</tr>
<tr>
<td>UCF</td>
<td>University of Central Florida</td>
</tr>
<tr>
<td>VHDL</td>
<td>VEHSEC Hardware Description Language</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>2D</td>
<td>Two Dimensional</td>
</tr>
<tr>
<td>3D</td>
<td>Three Dimensional</td>
</tr>
</tbody>
</table>
1 STRUCTURED DESIGN METHODOLOGY FOR SFF

A structured design methodology played a key role in the VLSI (Very Large Scale Integration) revolution. In this methodology, the design is abstracted at different levels (viz. system, function, logic, circuit, and layout) taking into account concerns relevant to a particular level of abstraction. This allows development of CAD tools that can describe, synthesize, and simulate the design at high levels of abstraction independent of the details of the lower levels.

The multilevel design paradigm is possible only if the interface between two successive layers of abstraction can be specified in a precise and clean way. The key observation that led to a clean interface between design and fabrication was the discovery of a set of scaleable design rules at the layout level allowing processing steps to be defined independent of an object’s geometry. Except for the system level, earlier research on digital design had already established clean interfaces between other levels. With the advent of high-level hardware description languages like VHDL, the spectrum of design abstraction has been extended all the way to the system level [Mead80], [Conway81].

In recent years, many researchers have observed that the lessons learned from the “VLSI experience” could be profitably applied to new classes of manufacturing processes called SFF (Solid Freeform Fabrication) and MEMS (Micro-ElectroMechanical Systems). This thesis is specifically concerned with the development of a structured design methodology for SFF processes.

SFF is a layered manufacturing technology in which a three dimensional structure is decomposed into thin cross-sectional slices or layers built one on top of the other, embedded in complementary shaped support structures if necessary. The fabrication process is thus pattern-insensitive, like that of VLSI. Several SFF processes exist and are being developed at private companies and research laboratories. Some examples of these technologies are stereolithography, laminated manufacturing, 3D printing, shape deposition manufacturing and laser-aided processes such as Laser Aided Direct Rapid Prototyping (LADRP) developed at the University of Central Florida, and Selective Laser Sintering (SLS) developed at the University of Texas, Austin[NSF95]. The last two laser aided processes are of special interest to this work and will be discussed briefly in the next section.

The advent of these new technologies, which share with VLSI the layering paradigm, have opened up new possibilities for rapid prototyping and agile distributed manufacturing. The key research questions that have been posed by these developments are: are there design hierarchies similar to the VLSI design hierarchy that can be exploited to develop verification and synthesis tools for SFF processes? What are the languages and interfaces between the layers of abstraction? What kinds of data structures are suitable to describe three dimensional geometry and what are the basic algorithms to generate layers, verify design rules, and generate process plans? This thesis is concerned with developing algorithms for slicing starting from a 3D polyhedral solid and generating the process plans for the LADRP and SLS processes. Specifically, the main contributions made in this thesis are:

1. Development of a direct slicing algorithm for arbitrary polyhedral solid objects based on a new plane sweep technique. The plane sweep method is adapted from computational geometry algorithms to work with the data structures in University of California Berkeley Unigrafix software which uses a BREP representation for solids.
2. We present two process planners capable of taking the layers generated from the slicer and producing output in the format of commands to the LADRP and SLS process controllers.

3. Development of a new geometric algorithm for labeling slice contours into a clockwise (CW) or counter-clockwise (CCW) ordering based on the geometry of the slice.
2 CONTRIBUTING TECHNOLOGIES

2.1 Laser Aided Direct Rapid Prototyping

Most of the widely used SFF processes are two-step indirect processes. Typically, the first phase is used to prepare a mold; the part is then produced by pouring molten material into this mold. Another method involves the joining of materials by some bonding substance, and a sintering technique is then used to solidify the bound structure. Under an NSF grant, a new Laser Aided Direct Rapid Prototyping system is now being developed at the Center for Research and Education in Optics and Lasers (CREOL). This process is a one-step direct process to fabricate three dimensional parts using a laser beam. With all processes, there is a certain amount of post-processing required. This actually adds a third step, but since it is common to all processes, it is often discounted. This final phase involves tasks such as polishing, applying finishes, etc., and are not of interest to this research.

A typical LADRP setup is shown in Figure 1. A solid to be built by the LADRP system is designed and converted into appropriate control inputs on the Computer. These inputs are forwarded to the Controller, which continuously controls the material feed rate of the Feeder Unit, the position and velocity of the XYZ Stage, and whether the laser beam is on or off. The multi-axis XYZ Stage moves through 3D space in accordance with the Controller's commands. The laser beam is kept stationary and is focused at the spot where material is to be deposited via the melting process. The beam is turned on to deposit material and is turned off to stop depositing; each layer may require the laser to be turned on and off several times. After one layer is deposited, the workstage drops down by the thickness of the next layer and is positioned to start at a point determined by the geometry of the object to be produced.

The research described in this thesis focuses on the Computer node in this system, which executes both our Slicer and Process Planner software. Our Slicer develops a geometric description of each layer, and our Process Planner generates the low level control signals to turn the laser beam on or off, set appropriate feed rates, and control the movement of the stage. The Slicer and LADRP Process Planner algorithms will be described in Chapters 3 and 4 respectively.
The output of the LADRP Process Planner is a series of ASCII strings that represent commands to the laser controller. These strings are currently stored in a text file which is then read by the laser controller. Below is a typical representation of the output generated by the LADRP Process Planner. The object represented in Figure 2 is a cube with edges of length 2.0 units, as shown in Figure 2. The Slicer and Process Planner are not concerned with actual real world measurements, so all dimensions are represented using a generic measurement of “units”. With respect to the Process Planners, each unit translates to a centimeter.

Due to the large size of the output file (this one was several thousand lines) only a few iterations of the planner’s output are shown. The (#X) entries in the output file are not part of the LADRP syntax; they merely serve as bookmarks into Figure 2 below.
Figure 2: Cube with edges length 2.0
LADRP Process Plan syntax

penup   turns the laser off
start_plot   activate the system
# set_dimension   initialize the stage to work in either 2D or 3D
R feedrate   sets the rate R at which the powder is fed to the laser
penup   turns the laser off
pendown   turns the laser on
XYZ uline   moves the stage X units along the X axis, Y units along the Y axis, and Z units along the Z axis

Sample LADRP Process Plan file

start_plot
3 set_dimension
0.15 feedrate
0 penup
-1.00000 1.08000 0.0 uline (#1)
0.0 -0.08 0.0 uline (#2)
0 pendown
0.15 feedrate
2.00000 0.0 0.0 uline (#3)
0 penup
0.0 -0.08 0.0 uline (#4)
0 pendown
0.15 feedrate
-2.00000 0.0 0.0 uline (#5)
0 penup
0.0 -0.08 0.0 uline (#6)
0 pendown
0.15 feedrate
2.00000 0.0 0.0 uline (#7)
0 penup
0.0 -0.08 0.0 uline (#8)
0 pendown
0.15 feedrate
-2.00000 0.0 0.0 uline (#9)
0 penup

The width of the material being deposited for this example is 0.08 units. This sample initially moves the stage to point the laser at position (-1.0, 1.08). From there the stage is moved
in the negative Y direction to (-1.0, 1.0), and the laser is turned on and deposits a line of material from the above position to (1.0, 1.0). The stage moves in the negative Y direction 0.08 units, and deposits in the negative X direction to (-1.0, 0.92), and then repeats this cycle again. Four lines of a layer are deposited, each line being 0.08 units wide.

![Diagram of LADRP process](image)

Figure 3: Sample LADRP slice file

2.2 Selective Laser Sintering

SLS is another SFF process for fabricating 3D objects. A sample setup is shown below in Figure 4. It consists of a container that is able to move downward with successive layers; the container is large enough to hold the completed object. A roller distributes a layer of powder at the current level of the container. A laser is then directed onto the areas of the powder to sinter (fuse) the particles into a solid region of the completed object. All excess powder is then removed, the container level drops, and a new layer of powder is applied.

As with LADRP, a solid object is initially designed and processed by our Slicer software (the same slicer is used for both LADRP and SLS). We have developed an SLS Process Planner which generates the contour information that controls the areas to be sintered. Our SLS Planner algorithm will be described in Chapter 5.
An SLS process plan is an ASCII text file containing vertices ordered together into contours. It is the ordering of the vertices that is important to the SLS laser controller. For all solid sections of a layer, the vertices are listed in a counter clockwise (CCW) order, while holes in a layer are represented by listing the vertices in a clockwise (CW) order.

SLS Process Plan syntax

- **contour Z** this indicates a contour of an object at height Z
- **loop P** this indicates that the contour consists of P vertices
- **endloop** indicates the end of this loop in the contour
- **endcontour** indicates the end of the contour
- **X Y Z** gives the (X,Y,Z) coordinates of a vertex in the contour

It is important to note that the order of contours within a layer is not important. However, the ordering of vertices into CW or CCW, and the order of the layers must be maintained.

The sample file below represents two layers of the object in Figure 5 below. The first contour specifies the outer, solid section of the cube with edges of length 2.0 units. The second contour specifies the vertices of the interior cubic hole with edges of length 1.0 unit. The third and fourth contours represent the next layer of the object, with the same dimensions.
Figure 5: Cubic object with cubic hole

Sample SLS Process Plan file

```
contour 0.00000
loop 4
-1.000000 -1.000000 0.000000
1.000000 -1.000000 0.000000
1.000000 1.000000 0.000000
-1.000000 1.000000 0.000000
endloop
endcontour

contour 0.00000
loop 4
-0.500000 -0.500000 0.000000
-0.500000 0.500000 0.000000
0.500000 0.500000 0.000000
```
0.500000 -0.500000 0.000000
endloop
endcontour

contour 1.000000
loop 4
-0.500000 -0.500000 1.000000
-0.500000 0.500000 1.000000
0.500000 0.500000 1.000000
0.500000 -0.500000 1.000000
endloop
endcontour

contour 1.000000
loop 4
-1.000000 -1.000000 1.000000
1.000000 -1.000000 1.000000
1.000000 1.000000 1.000000
-1.000000 1.000000 1.000000
endloop
endcontour
2.3 Geometric Modeling

In order to formulate algorithms capable of manipulating 3D objects, we first need data structures that are capable of accurately representing these objects. The three primary modeling techniques are wireframe, surface, and solid models.

2.3.1 Wireframe Models

A wireframe model is the simplest, but most verbose geometric model that can be used to represent an object. It is an edge representation that generally consists of straight lines, circular arcs, conics, and B-splines.

Some advantages of the wireframe model are: the ease with which models can be created, and the basic nature of the arithmetic operations used on wireframe models. The major disadvantage is the ambiguity and incompleteness inherent in the method [Mantyla88]. Complex models such as the object in Figure 6 can be confusing and non-intuitive when inspected visually or mathematically. Multi-edge models will contain a mixture of interior and exterior edges, and when the appropriate interior edges are "hidden", the model becomes clear. However, the simple data structure used for wireframe models does not contain any of the geometric or topological information, such as visibility and shading, that are required by hidden line algorithms for processing. Thus it becomes necessary for the user to manually hide interior lines in the model in order for it to be clear; a process which can be very time consuming and error prone. Wireframe models are useful for a set of very restricted problems such as collision detection, but for more complex tasks the model does not provide the information necessary.
2.3.2 Surface Models

The surface model representation uses bounding surfaces and edges of a specified object, and also provides information on the surfaces connecting the object’s edges, but without any information on the ordering of the surfaces. Surface modeling is based on quadric surfaces, interpolations and approximations of surface patches such as Bezier and B-spline surfaces [Faux79].

The advantage of surface models is the ability to represent complex shapes. The major disadvantage is the extreme difficulty involved in point-set classifications (is a point in, on, or outside a surface) due to lack of topological information (such as the Cartesian coordinates of a point). Without this information it becomes necessary to perform conversions to and from the parametric format which surface models use. Since point-set evaluation is required for tasks such as calculating the union, intersection, and difference of various models, this limitation can be troublesome. Figure 7 below illustrates a basic solid, and a possible method in which the model could be stored as a surface model.
Figure 7: Surface model representation of a pyramid
2.3.3 Solid Models

A solid model provides a more complete representation than the previous two methods. In addition to providing the geometric information for a model, this method also gives the topological data (i.e., the relational information between each of the model’s pieces). This completeness in representation is necessary for any level of model manipulation to provide accurate, dependable results. The two primary methods of solid model representation are Constructive Solid Geometry (CSG) and Boundary Representation (BREP).

2.3.4 Constructive Solid Geometry

A CSG solid is represented using a binary tree. Each leaf is a primitive solid and each internal node is a regularized Boolean operation on its subtrees. A sample basic set of primitive solids could include cubes, cones, and tori. The set of available regularized operations is union, intersection, difference, complement, and rigid motions. The result of a regularized Boolean operation is a regular set, which is defined as follows: a regular set represents the closure of the interior of a set, i.e., \( A \cup^* B = \text{closure}(\text{interior}(A \cup B)) \). This closure has the effect of removing any lower dimensional entities, thus producing a complete solid (by lower dimensional entities, we mean points, lines, and planes). Figure 8 below illustrates the difference between set-theoretic and regularized Boolean operations on two objects [Requicha85].

\[ \text{Figure 8: Regularized Boolean operations} \]
Since the Boolean operations are performed on primitive solids, a CSG solid must be a bounded regular set, i.e., the set can be formed using concatenation, union, and closure in an arbitrary order on the set of primitives. If the CSG solid is not bounded and regular, the regularized operations cannot produce a closed set, and the model will be invalid. If the set of primitives in the CSG solid is bounded, the regularized operation guarantees that its operations will produce an algebraically closed model (i.e., no lower dimensional entities) [Mantyla88].

Figure 9 below represents a CSG tree with two primitives and a regularized Boolean operation, difference.

Figure 9 : CSG tree of an object

2.3.5 Boundary Representation

BREP modeled objects are graphs whose nodes represent faces, edges, and vertices, with the links between these nodes representing the adjacency relations between each of the nodes. Boundary based representations contain information about the surfaces of the individual objects that make up a complete solid. Each surface is fully described along its own boundaries by using two dimensional entities such as edges and vertices. The adjacency relationships between each of these surfaces is what forms the subsequent three dimensional volume. Section 2.3.7 discusses a sample BREP data structure, the Winged Edge structure.

The topological information provided by a BREP model is different from that of CSG or surface models. It provides two major items unique to BREP: numerical values of all lower dimensional entities, and the set of adjacent entities for each point, edge, and face. This produces
specific numerical values for every part of a BREP model, enabling the user to query and manipulate the model using actual numerical values.

Figure 10 below shows the BREP equivalent of the CSG tree in Figure 9 above. Notice that the BREP object consists only of 2D planar objects, that when joined together form the boundaries and faces of the 3D model.

Figure 10: BREP representation of an object
2.3.6 Associated Issues

Presented below are a few of the issues involved in the use of CSG and BREP for solid models; refer to [Bannerjee95] for a more in depth discussion of these issues. The greatest advantage of the CSG model lies in the fact that the validity of the model is guaranteed, i.e., each primitive within the solid is guaranteed to be a complete solid by itself. Using the primitives and Boolean operations creates a well-defined solid object with known properties. A major problem with the CSG representation is that the topology CSG provides does not give all the algebraic and adjacency information needed for model manipulation. CSG primitives are complete solids, but the representation of these solids does not hold needed algebraic information such as the parametric equations of the lines and planes making up the edges and surfaces of the primitive [Mantyla88]. Also lacking from the model is information such as edge and face adjacencies, and the ability to evaluate an object at its boundaries. This information can be obtained from a CSG model by converting it into BREP. If left strictly in CSG format, the algebraic topology information is not available. These weaknesses in CSG are the primary strengths of the BREP model. The numerical values of the vertices, edges, and faces that make up the boundaries of the BREP object are readily available, thus allowing for the manipulation of objects based on their geometry. This property of BREP models is of key interest to this research as it allows the possibility of breaking a model down into layers.
2.3.7 Winged Edge Representations

There are many BREP representations; an important one is the winged edge representation. In a winged edge structure, each face is represented by a list of edges that is traversed in a clockwise order. In order for a solid to have a complete definition, each edge must be a part of two adjoining faces. Thus each edge is traversed twice, once in the positive (CW), and once in the negative direction (CCW) [Mantyla88]. By using the information in the winged edge structure, the connectivity of every point, edge, and face in the solid is known, thus providing the ability to manipulate and retrieve the algebraic information discussed in section 2.3.6 from the model easily. Every winged-edge structure is of the form $WE = (V, E, F, CW, CCW)$, with $V$ being the set of all vertices in the model, $E$ the set of all edges, $F$ the set of all faces, $CW$ being the set of pairs clockwise edges corresponding to a particular edge, and similarly $CCW$ the set of all counter-clockwise edges. As an illustration, Figure 11 below shows a graphical representation of the winged edge structure with respect to a single edge in the model. Each field in the figure is defined as follows:

$vstart$ & $vend$ : the terminating vertices of an edge
(from $V$ and $E$)

$ncwe$ : the next clockwise edge of a face where the edge is traversed in a positive orientation
(from $E$, $F$, and $CW$)

$nccwe$ : the next counter-clockwise edge in the adjoining face
(from $E$, $F$, and $CCW$)

$pcwe$ : the preceding clockwise edge
(from $E$, $F$, and $CW$)

$pccwe$ : the preceding counter-clockwise edge
(from $E$, $F$, and $CCW$)

$few$ & $fccw$ : identifiers of the faces neighboring the edge
(from $E$ and $F$)
2.4 Unigrafix

Unigrafix is a software package developed at the University of California, Berkeley. The package was initially developed as separate functional modules by undergraduate and graduate students in the mid 1980's as a series of class projects.

The initial intent of Unigrafix was to provide simple graphics and modeling capabilities for 3-dimensional polyhedral objects. The software is portable across several UNIX platforms, and consists of a descriptive BREP language and several programs that allow the user to create, modify, and display scenes consisting of 3D polyhedral objects. If desired, further information on Unigrafix can be obtained from [Sequin83] and [Sequin90].
2.4.1 Unigrafix Data Structures and Syntax

Unigrafix has a single core data structure providing the basis for the entire system: a modified winged edge representation consisting of vertices, edges, and faces. Each of these primitives is defined by Unigrafix as follows:

**vertex**: consists of 3 floating point numbers representing the X, Y, and Z coordinates of the vertex.

**edge**: consists of two pointers, one to each vertex that makes up the edge, and pointers to any edges that are adjacent to it.

**face**: consists of pointers to each edge that is a part of the face; the edges are pointed to in an order that indicates which side of the contour is the outer part of the face. It will also contain a series of pointers to all other face objects it is connected to.

The above information provides all the needed functionality in order to define complete BREP models. However, the use of floating point numbers can create well known problems. The more the model is manipulated, the more likely it becomes that information will be lost due to round off error.

All input to Unigrafix comes from text files. There is a range of commands available to users for the creation of solid models. Below are the commands that are of specific interest to this thesis.

**vertices**: create a vertex with the identifier ID and the 3D floating point coordinates x, y, and z. Unigrafix does provide the capability to use homogeneous coordinates, but this was not of use to this research so is not discussed here.

\[ v \ ID \ x \ y \ z; \]

**wires**: create a “wire” (an edge) with the identifier ID, connecting the given vertices with ID’s vl, v2, ..., vn in the specified order.

\[ w \ ID \ (vl \ v2 \ .. \ vn); \]

**faces**: creates a face with identifier ID, connecting the ID’s vl, v2, ..., vn in the specified order; the input ID’s (vl..vn) can be either vertices or edges.

\[ f \ ID \ (vl \ v2 \ .. \ vn); \]

**instance**: declare an object with identifier ID of the type defID, where defID is a previously defined object.

\[ i \ ID \ (defID \ [transforms]); \]

**include**: functions like a C header file, allowing objects to be defined in other files and then declared in another file as an instance.
Sample file #1

This sample file represents a cube \( A \), with dimensions \( 2 \times 2 \times 2 \), with a rectangular hole \( B \), inside with dimensions \( 1 \times 1 \times 2 \), as illustrated in Figure 11 below.

```
#include filename [transforms];
```

Figure 12 : Sample Unigrafix file, Cube with hole

```
/* SAMPLE FILE #1 */
/* define the points in cube A */
v XYZ 1 1 0;
v XZ 1 -1 0;
v YZ -1 -1 0;
v Z -1 1 0;

v XY 1 1 2;
v X 1 -1 2;
v Y -1 -1 2;
v N -1 1 2;

/* define the points in hole B */
```
v ll .5 .5 0;
v mm .5 -.5 0;
v nn -.5 -.5 0;
v oo -.5 .5 0;
v pp .5 .5 2.0;
v qq .5 -.5 2.0;
v rr -.5 -.5 2.0;
v ss -.5 .5 2.0;

/* define the faces of cube A */
f base ( XY X Y N );
f top ( XYZ XZ YZ Z );
f z ( XYZ XY N Z );
f a ( XZ X Y YZ );
f b ( XYZ XY X XZ );
f c ( Z N Y YZ );

/* define the faces of hole B */
f yy ( pp qq rr ss);
f xx ( ll mm nn oo);
f ww ( ll pp ss oo);
f vv ( mm qq rr nn);
f uu ( ll pp qq mm);
f tt ( oo ss rr nn);

Sample file #2

This file includes two objects defined externally to this file and merges them into a single object.

/* SAMPLE FILE #2 */
def slc1;
    include slc1;
end;

/* include the contents of file slc1 in this object */
i A ( slc1 );
def slc2;
    include slc2;
end;

/* also include the contents of file slc2 in this object */
i B ( slc2 );
2.5 Design Hierarchy

As noted earlier, a structured design methodology based on multi-level design abstraction and a digital interface that separated the design process from fabrication contributed much to the unprecedented success of VLSI technology. For mechanical and electromechanical systems, in general, finding a multi-level design hierarchy is a difficult challenge because the underlying models involve energy transformation and physical parameters; the elemental components share function and behave differently in a system due to back loading [Voelker94][Whitney94].

For the family of SFF processes, at least the following four levels of design hierarchy can be identified:

- **Design level, including function, features, and properties.**
  At this high level, a formalism is needed to capture the functional behavior of the mechanical system from the physical parts without specifying the geometric information. Relevant mechanical and physical properties of the material in the context of the intended application must be taken into account at this level. The definition of a standard interface at this level to the 3D geometry level below it will have to wait until lower level languages and interfaces are better understood.

- **Three dimensional geometry level**
  At this level, a completely process independent representation of the system in terms of its geometrical shape and material in 3D has to be specified. We will call it a design subsystem which performs the traditional design by deriving the shape and geometry that achieves the desired functional specification. The design must satisfy a set of design constraints with respect to a set of relevant mechanical and physical properties of the material. The analysis and simulation tools verify the correctness of the design taking into account material strength, volumetric and surface properties, and thermal and fluid flow properties if necessary. The design may go through multiple refinements before being delivered via the 3D digital interface to the physical design layer [Sproull94]. The design languages to be used for data exchange at the digital interfaces are areas of active research.

- **2.5D layered level adapted to specific SFF technology**
  The physical design phase uses specific knowledge of the process and its design rules to specify a 3D description of the part. Ideally, like in VLSI which satisfies the layering paradigm with conservative design rules, this stage should be insensitive to the object's geometry. As is well known from studies of SFF processes, such a layering paradigm does not work in practice for objects with undercut, objects with multi-graded material, or with material having anisotropic density. Furthermore, the smooth 3D surface features have to be compromised by linear approximation of zigzag surface features, and additional constraints are required if the object needs support structures or overhangs.
  The translation of the 3D geometry to 2.5D geometry has to be done by a slicer that will produce the 2.5D layers, given its description in SIF. A language called Layered Solid Interchange Format (L-SIF) has been suggested for this layered description, but no specifics have been provided for any SFF processes [Sequin95][NSF95]. This description will form a 2.5D digital interface between the physical design and the process planning stage. The development of a
standard L-SIF language for all SFF processes will be a difficult challenge at this stage because the SFF technologies are varied and are still in the process of development and maturation. There are no standard SFF processes like the standard CMOS process in VLSI. University/Industry cooperation in setting up experimental research testbeds for standard SFF processes might lead to development of such standard L-SIF languages.

- Process planning and fabrication
This is the final stage of the hierarchy at the lowest physical level. If the design is validated by simulation and is free from both design constraints and design rules errors, the design is sent down to the process planning stage which generates the information for automatic sequencing of operations for the particular SFF process. In our case, the process in question is LADRP. Figure 13 shows the design hierarchy we are interested in.

![Design Hierarchy Diagram]

Figure 13 : Design Hierarchy

These suggested “design levels”, as well as other hierarchies being researched can be found in many publications. Some of the more noteworthy ones that were beneficial to this research were [Chern94], [Conway81], [Kar96], [Mead80], [Merz94], and [Mukherjee94].
3 The Slicer

3.1 General Algorithm Description

The goal of this algorithm is to decompose a solid model into "slices" that can be suitably described to a SFF system for manufacturing. Since models are designed in 3D, it becomes necessary to have a simple method by which a model can be broken down into 2D layers [Franceschini97]. This lead to the major purpose of this research, the development of a simple and robust slicer.

A formal definition of the slicer is as follows: given a 3D object described in BREP format, an axis to slice along, and slice sizes as input parameters; the slicer will be able to remove a 2D cross sectional slice from the model along the given axis while maintaining the integrity (a proper BREP definition) of the remaining 3D model and the 2D slice. Given that all generated slices maintained their integrity and that of the model, the slices can be rejoined in order as layers, and the resulting 3D model should be identical to that provided as input to the slicer.

Intuitively, the slicer implemented in this thesis uses a "space-sweep" algorithm. The main idea is to sweep a plane oriented parallel to either the yz, xz, or xy plane (i.e., along the x, y, or z axis respectively) through the solid model of the object. The plane stops at event points along the axis for processing. A sweep data structure is maintained that provides local information about the portion of the solid that has been swept and is contributing to the current slice.

The inputs to the algorithm include the solid model (in Unigrafix BREP format), and the axis (A) upon which to sweep, and the width of each slice to be taken from the object.

In the first stage of the algorithm, the event queue is constructed using the solid model and the current slice width. Two types of events can then be added to the event queue: vertex events and slice events. Events are placed on the queue in increasing order of their corresponding values along the A axis (ordering of multiple events with the same A axis value is described below).

Vertex events are added to the event queue for each A axis value corresponding to a vertex in the solid model. If more than one vertex shares an A axis value, then the event maintains a list of all vertices involved in the event.

Slice events are added to the event queue at the A axis value at which the slice will occur in the solid model. Let \( \alpha(i) \) denote the A axis value of the sweep plane at event \( i \).

\[
\alpha(i) = \alpha(i-1) + \text{slice_width}(i)
\]

If a vertex event and a slice event occur at the same A axis value, then the vertex events are given priority and will occur first.

In the second stage of the algorithm, the plane is swept along the A axis, stopping at each event point. A sweep data structure containing a BREP model of the current slice of the solid is built as the plane sweeps through the solid. The processing at each event point is summarized below.
Vertex Event: In this case, all vertices at this A axis value are added to the sweep data structure along with any edges of the solid incident to these vertices that are not already within the slice model.

Slice Event: In this case, the sweep plane is intersected with the sweep data structure by intersecting it with the appropriate edges in the structure. The resulting “slice” is then output by the algorithm. The top of the slice (the part cut by the sweep plane) is stored as the base of the next slice, and the vertices within the sweep structure that fall below the sweep plane are removed along with any edges entirely below the sweep plane.

3.2 Slicer Algorithm

The algorithm takes the following inputs: the original solid model, the axis (x, y, or z) along which to slice, and the width of each slice (which, when combined with the previous two pieces of information implies S, the number of slices). There are four important data structures used by this algorithm: the original input solid model represented using a set of winged-edge data structure \( M = (V_M, E_M, F_M, CW_M, CCW_M) \), a priority queue \( Q \) representing the event schedule, and two more winged-edge data structures \( W = (V_W, E_W, F_W, CW_W, CCW_W) \) representing the sweep data structure, and \( O = (V_O, E_O, F_O, CW_O, CCW_O) \) representing each output slice as it is generated. Each of the individual edges of the above models is represented using the representation specified in section 2.3.7. The events in \( Q \) are vertex and slice events arranged in ascending order according to their slice axis values. There are at most \( |V_M| \) vertex events and exactly \( S \) slice events in the schedule.

Given the above information, our slicer algorithm is specified as follows:
while( Q \neq \emptyset )
{
    get the top element of the queue
    if the event is a vertex event then
        for each vertex in the event
            Insert the vertex and all its connected edges into W.
    }
    else a slice event has occurred
    Partition W into two solids, W' and O; where W' contains the portion of W above
the sweep plane and O contains the solid portion of W below sweep plane

W = W'

output the slice model O to the process planners
}

3.3 Slicer Implementation

The slicer operates on a single key data structure which is a derivative of the Winged
Edge data structure discussed previously. This structure is used to hold all the BREP information
needed to accurately represent the 3D model and the 2D slices generated from the model.

The structure contains two linked lists: one of vertices, and one of edges. A vertex entry
consists of the (x, y, z) coordinates of the vertex, and pointers to each edge that the vertex is a
member of. An edge entry contains two pointers, one to each vertex making up the edge.

The data structures

SLICE_STRUCT
{
    VERTEX_ENTRY *vertices;
    EDGE_ENTRY *edges;
}

VERTEX_ENTRY
{
    char *name;
    float64 coords[3];
    EDGE_ENTRY *edges;
    VERTEX_ENTRY *next_point;
}
EDGE_ENTRY
{
    VERTEX_ENTRY *point_1;
    VERTEX_ENTRY *point_2;
    EDGE_ENTRY *next_edge;
}

The algorithm implementation consists of the same two primary parts as the "space-sweep", (i.e., the scheduler and the slicer). The scheduler's primary use is to define the stopping points along the slice axis at which the sweep plane must stop. The slicer's job is to calculate the intersection of the sweep plane with the 3D model and determine the 2D structure of the layer being removed.

In the first pass of the algorithm, the scheduler processes the object and identifies the event points for the model. Two significant types of events are derived from the model: a new vertex event and a slice event. Each vertex in the model will occur in only one vertex event, and slice events are scheduled based on the length of the object along the slice axis, and the width of the slices to be used as indicated by the adaptive input. Every event is sorted in ascending order based on the location at which it occurs along the slice axis. Two events can be scheduled at the same time, and the method of resolution depends on the nature of the events.

Vertex & Vertex: if multiple vertices reside in the same plane along the sweep axis, then the vertices will all share a common vertex event and will be processed together.

Vertex & Slice: vertex events will always be given priority over slice events since vertices that occur on the slice plane become part of the slice.

Slice & Slice: this should not occur.

At a vertex event all vertices sharing that event are added to the current 2D slice, along with all edges connected to each vertex. Since the slice structure is emptied after each slice event, each new vertex event implies that the point is a part of the new slice.

The slice event is used to determine the intersection of the sweep plane with the edges in the 3D model. Once the slice event has been completely processed, the resulting 2D slice is passed as input to the process planners. The actual implementation follows this general method.

For this discussion, assume we are sweeping along an axis denoted as A, and the value of the sweep plane at event i will be \( \alpha(i) \). The "floor" of a slice thus becomes \( \alpha(i-1) \), and the "ceiling" becomes \( \alpha(i) \). The slice is represented by the portion of the 3D model that falls within the floor and ceiling planes. Each edge in the model must now be checked to determine where it falls with respect to the slice boundaries. Given the above assumptions, this check is based on the A coordinate values of the two points making up the edge. This check can yield three possible results:
1) the entire edge is inside the slice, or
2) the entire edge is outside the slice, or
3) part of the edge falls within the slice.
Figure 14 below illustrates each of these conditions.

![Hexagonal object with sweep planes](image)

**Figure 14**: Hexagonal object with sweep planes

In case #1, the edge is added to the slice structure with no additional processing. In the above figure, this would include edges GH and DC. In case #2, the edge is disregarded as it does not affect this slice, again no more processing needed on this edge. In the above figure, this would include edges FE and AB. Case #3 is more involved. First it must be determined which (or both) of the two vertices making up the edge fall outside the floor/ceiling boundaries. For each vertex falling outside the slice, the intersection of either the floor or ceiling plane with the edge is calculated using standard parametric equations. The points generated from this intersection check are then added into the slice structure as new vertices, along with a new edge connecting them. From the above figure, this would include edges GF, AH, BC, and ED.

Below is the algorithm used in our slicer implementation.

**Slicer Algorithm**

\[
L = \text{a sorted list of all vertex and slice events}
\]

while( number of events_processed < total number of events )
if the current event from L is a vertex event, then
    for all vertices in the vertex event
    { insert the vertex into the current slice structure }
else a slice event has occurred
    while( untraversed edges exist in the slice structure )
    { set the current edge = the next untraversed edge
        calculate the intersection of the current edge with the sweep plane
        if the edge intersects with the plane, then
        { store the resulting point of intersection and the newly created
            edge in the slice structure }
    }
output the slice to the process planners
    set the sweep plane to its next value
}
3.4 Slicer Algorithm Analysis

3.4.1 Algorithm Analysis

The formal algorithm breaks down into three significant parts: generating the event schedule, updating the sweep structure, and calculating the intersection of the sweep plane with the 3D model.

The event queue can be built as a balanced binary tree, (e.g., 2-3 tree or an AVL tree), an operation with a well known complexity of \(O(N\log N)\), where \(N \leq (|V_M| + S)\) is the number of events\[Naps86\], \(S\) being the total number of slice events in the model. Therefore the construction of the event queue takes \(O((|V_M| + S)\log(|V_M| + S))\) time.

Updating the sweep structure consists of inserting a vertex and its adjacent edges that lie above the sweep plane into the sweep structure. Given that \(L, \deg(v) = 2\epsilon \epsilon M\), and we have \(|V_M|\) vertices, the average degree of a vertex is \(2|E_M|/|V_M|\). Because a winged-edge structure is a planar graph, we know that \(|E_M| \leq 3|V_M| - 6\), so \(|E_M| = O(|V_M|)\). Thus updating the sweep structure for each vertex is an \(O(1)\) operation.

The final portion, intersection calculation, consists of intersecting the sweep plane with every edge in the slice structure. As a worst case, \(|E_M|\) edges could be in the structure, thus yielding a complexity of \(O(|E_M|)\).

While the first part is only executed once, part two is run for each insertion event, or \(O(|V_M|)\) times, and part three must be run for each slice event, or \(O(S)\) times. This produces an overall algorithm complexity of: \(O((|V_M| + S)\log(|V_M| + S) + |E_M| + |E_M|S)\). Given that in most models, \(S\) will be much greater than \(|E_M|\), the above value is dominated by the \((|V_M| + S)\log(|V_M| + S)\) factor.

3.4.2 Implementation Analysis

The slicer implementation has the same three significant sections as the algorithm presented above. The only real difference being in the first section, the creation and sorting of the event queue.

For the implementation, the construction of the event schedule is done with a standard insertion sort, with a well known complexity of \(O((|V_M| + S)\log(|V_M| + S))\) \[Naps86\]. The decision to used the insertion sort was based on two things: the ease of coding, and the fact that in general the queues will not be of a size large enough that the complexity will become an issue. So in contrast to the above algorithm, the implementation has an overall algorithm complexity of: \(O((|V_M| + S)^2) + |E_M| + |E_M|S)\), which is dominated by the \((|V_M| + S)^2\) factor.

3.5 Floating Point Error

Throughout Unigrafix, floating point values are used to represent the coordinates of the vertices. The errors associated with floating point numbers are well known, and will not be discussed here. However, their use does bring up the possibility of error in the slicer. This
creates two problems: do the generated 2D slices accurately represent the original 3D model, and will the output to the laser processes be precise and consistent enough for the fabrication of the model.

If any error was occurring in the slicer, it would be occurring on the slice boundaries as this is where actual floating point calculations are made to create intersection points. This would create gaps between the 2D slices when they are merged. It would also lead to faulty fabrication of the model when sent to the Process Planners.

To this end a small test was devised. The test consists of storing the values of all points on the floor of the current slice, and comparing them against the value of all points on the ceiling of the previous slice. The points should be returned as identical, or else a gap will occur between the two layers. This test was run for a precision of up to six places past the decimal point, more than enough significance for the laser processes. The percentage of error was exactly 0.0%. No error was registered in any runs of the slicer, on any form of object.

These results might seem a little unrealistic, but they are easily explained. The same value is used for all points throughout the run, from the initial read and store from the data files, through all slice operations, and throughout the output procedures. This provides a set of stable endpoints for every edge so that when intersection calculations are performed, they are always performed with the same values, using the same parametric equations. Thus the points generated for the floor of a slice will always be identical to the points in the ceiling of the previous slice.

Below is the algorithm that was used to determine the amount of error in the slicer.

**Floating Point Error Check Algorithm**

Given two slices $A$ and $B$, where $B$ is the slice created immediately after $A$, there exist the following sets of points:

- $C(A)$ - set of all points on the ceiling of slice $A$,
- $F(B)$ - set of all points on the floor of slice $B$,
- $C(B)$ - set of all points on the ceiling of slice $B$.

RANGE - a user set value to for testing the floating point accuracy defaults to $0.00001$

for all slices generated from the model
{
    for every vertex $i$ in $C(A)$ and $C(B)$
    {
        if vertex $i$ of $C(A)$ is not within RANGE of vertex $i$ of $F(B)$, then
            an error has occurred and the slicer results are no longer valid
    }
    set $C(A) = C(B)$
}
4 Process Planning

4.1 Laser Aided Direct Rapid Prototyping (LADRP)

The goal of the LADRP process planner is to produce a series of text commands that may be passed to a laser controller, thus causing the laser to build a part. In order to generate these commands, certain geometric properties of the slice must be determined. To do this, another modified version of the “space-sweep” algorithm is used. This modified version uses a sweep plane that runs perpendicular to the sweep plane used in the slicer (i.e., if the slicer used the z axis, the planner would use either the x or y).

The algorithm will again consist of vertex and slice events, with the vertex events being identical to those used above. However, this time the slice event will function more as a line connection algorithm, where the intersection points generated from the plane and the slices edges are used as line segments. These line segments are then ordered and used as positions at which the laser is to be turned on or off.

4.1.1 LADRP Planner Algorithm

As input, the Process Planner accepts a 2.5-D slice (without loss of generality) \( M = (V_M, E_M, F_M, CW_M, CCW_M) \), containing \( |V_M| \) vertices, and \( |E_M| \) edges, and a real number \( d \), representing the width of the laser material to be deposited. Other data structures of note include the priority queue \( Q \) representing the event schedule, a list of points, \( L \), representing the intersections of the plane with \( M \), and a winged edge structure, \( W = (V_W, E_W, F_W, CW_W, CCW_W) \), representing the sweep structure. The only events in the schedule are vertex events, and there will be at most \( |V_M| \) such events, all of which are arranged in descending order by their \( Y \) coordinate values. Given this information, the LADRP planner algorithm is shown below.
while \((Q \neq \emptyset)\) 
}\{ 
  if the current event is a vertex event, then 
  \{ 
    for each vertex in the event 
    \{ 
      Insert the vertex and all its connected edges into \(W\) in ascending horizontal order. 
    \} 
  \}

else a slice event has occurred 
\{ 
  Determine the intersection of the sweep plane with all edges currently in \(W\); store the resulting \(X\) values of the intersection points in the sweep structure 

For any edge that does not intersect with the sweep plane, remove the edge from \(W\), (it has fallen below the sweep plane) 

output the intersection points to the laser controller as a series of commands 
\}

get the next event from \(Q\)
\} 

4.1.2 LADRP Planner Implementation

In general, the algorithm will need to perform certain basic operations. These operations fall into two general categories: sorting and intersection calculation. At a minimum, an initial sort of the points and edges is required. Because of the line-by-line method in which the LADRP process creates a layer, intersecting a line with the edges of the structure is the simplest and most direct way of obtaining the information needed for the fabrication process.

Conceptually, the algorithm operates similar to a sweep line algorithm. A sweep line with the value \(\alpha\), where \(\min(Y\ \text{coordinate}) \leq \alpha \leq \max(Y\ \text{coordinate})\), is run through the edge list of the slice. Every intersection of the sweep line and an edge produces an intersection point, and the \(X\) coordinate value of the intersection point is stored. Once all edges in the slice have been checked for intersection, the stored intersection points are sorted into ascending or descending order using a standard insertion sort. The order of the sort depends on the direction that the laser is traveling, left to right, or right to left.

To have a valid object representation, the outer edge of the model must be the edge of a solid area. From this, it can be assumed that the first intersection point in the array is always on an edge of a solid. It is also known that the points must follow this pattern: solid, hole, solid, hole,.....,hole, solid. So the process issues the command to deposit from solid to hole, move
without deposit from hole to solid, and repeat until the last intersection point has been reached. Once a complete line has been deposited, the stage is moved in the negative $Y$ direction the width of the input material, and the process is repeated going the opposite direction. This is repeated until the entire layer has been deposited, at which point the stage is lowered and layer deposition may begin for the next slice.

**LADRP Process Planner implementation algorithm**

$L$: a list of $X$ coordinates of the intersection points from the current sweep plane

find the max and min $X$ and $Y$ values in the slice

move the stage to $(X_{\text{min}}, Y_{\text{max}})$ without depositing material

while( the $\alpha$ axis value of the sweep plane $\geq$ the minimum $Y$ value )

{ for every edge in the sweep structure
  { if the sweep line intersects with current edge, store the $X$ coordinate of the intersection point in the list $L$, and increment num_intersections. }
  
  if the laser is moving left to right, then
  { sort the list $L$ in ascending order }
  
  else the laser is moving right to left
  { sort the list $L$ in descending order }
  
  for every intersection point in the list $L$
  { if the point is on an even index ($i$), then it is a deposition start point, so
    { Move the stage, while depositing, from $L(i)$ to $L(i+1)$ }
  } else turn the laser off and move it to the next point
  { Move the stage, without depositing, from $L(i)$ to $L(i+1)$ }
  }

if the direction of movement is left to right, then
{ change direction of movement to right to left

41
Move the stage, without depositing, to $X_{\text{max}}$ and down by the width of the material being deposited

} else

{ change the direction of movement to left to right

Move the stage, without depositing, to $X_{\text{min}}$ and down down by the width of the material being deposited

}

4.1.3 LADR Planner Algorithm Analysis

The formal algorithm breaks down into three significant parts: generating the event schedule, updating the sweep structure, and calculating the intersection of the sweep plane with the 2.5-D model.

The event queue can be built as a balanced binary tree, (e.g., 2-3 tree or an AVL tree), an operation with a well known complexity of $O(N \log N)$, where $N \leq |V_M|$ is the number of event. Therefore the construction of the event queue takes $O( (|V_M|) \log(|V_M|) )$ time.

Updating the sweep structure consists of inserting a vertex and its connected edges into the sweep structure. Given that $\sum_{v \in V_M} \deg(V_M) = 2|E_M|$, and we have $|V_M|$ vertices, the average degree of a vertex is $2|E_M| / |V_M|$. Because a winged-edge structure is a planar graph, we know that $|E_M| \leq 3|V_M| - 6$, so $|E_M| = O(|V_M|)$. Thus updating the sweep structure for each vertex is an $O(1)$ operation.

The final portion, intersection calculation, consists of intersecting the sweep plane with every edge in the sweep structure. As a worst case, $|E_M|$ edges could be in the structure, thus yielding a complexity of $O(|E_M|)$.

While the first part is only executed once, part two is run for each insertion event, or at most $O(|V_M|)$ times, and part three must be run a variable number of times based on the width of the depositing material $d$ and on the total width of the slice $w$. The number of required iterations is $K$, where $K \leq \text{width}(w)/d$. This produces an overall algorithm complexity of: $O( (|V_M|) \log(|V_M|) + |E_M| + |E_M|K )$, which is dominated by the $|E_M|K$ factor.

4.1.4 LADR Planner Implementation Analysis

The formal algorithm breaks down into three significant parts: obtaining the max and min Y coordinate values, calculating the intersection of the sweep plane with the 2.5-D model, and sorting the resulting intersection list.
Given that the vertices are contained in an unsorted manner, the max/min values are determined by a simple linear search. Given \( |V_M| \) vertices, this produces a complexity of \( O(|V_M|) \).

The second portion, intersection calculation, consists of intersecting the sweep plane with every edge in the slice structure. For all cases, \( |E_M| \) edges will be in the structure, thus yielding a complexity of \( O(|E_M|) \).

The final portion, sorting the list of all intersection points is again an insertion sort with a well known complexity of \( O(N^2) \). As a worst case, every edge could be intersected, yielding a complexity of \( O(|E_M|^2) \).

While the first part is only executed once, while parts two and three are run a variable number of times based on the width of the depositing material \( d \), and on the total width of the slice \( w \). The number of required iterations is \( K \), where \( K \leq \text{width}(w)/d \). This produces an overall algorithm complexity of \( O(|V_M| + |E_M|K + (|E_M|^2)K) \), which is strongly dominated by the \( |E_M|^2K \) factor.

4.2 Selective Laser Sintering (SLS)

The goal of the SLS process planner is to produce a series of text commands that may be passed to a laser controller, thus causing the laser to build a part. In order to generate these commands, certain geometric properties of the slice must be determined. To do this, a modified version of the "space-sweep" algorithm will be used, along with another algorithm for the determination and classification of contours.

The SLS controller input is in the form of contours, which are in turn determined by a clockwise or counter clockwise ordering of the vertices making up the contour. This CW/CCW ordering determines whether the contour indicates a solid or hollow region of the model.

The initial pass of the SLS planner will be used to identify the contours as separate entities. The next pass is designed to use the modified "sweep-plane" algorithm to determine whether the contours represent solid or hollow regions. And the final pass is for the determination of the appropriate CW/CCW ordering of the vertices within each of the contours.

4.2.1 SLS Process Planner Algorithm Specification

As input, the Process Planner accepts a 2.5-D slice \( M = (V_M, E_M, F_M, CW_M, CCW_M) \), containing \( |V_M| \) vertices, and \( |E_M| \) edges. Other data structures of note include the priority queue \( Q \) representing the event schedule, a list of points, \( L \), representing the intersections of the sweep plane with \( M \), and a winged edge structure, \( W = (V_W, E_W, F_W, CW_W, CCW_W) \), representing the sweep structure that holds the labeled contours. The only events in the schedule are vertex events, and there will be at most \( |V_M| \) such events, all of which are arranged in ascending order by their \( Y \) coordinate values. Each edge in \( M \) must be classified into a contour, and each contour must then be labeled as either a hole or a solid.

Given the above information, the SLS planner algorithm becomes:

\[
\text{while( there are still unlabeled edges in the slice ) }
\]

\[
\{
\text{retrieve the next unlabeled edge in the model}
\}
\]
label the edge and all edges connected to it with a unique identifier and insert them all into W.

while( Q ≠ ∅ )
{
  Calculate the intersection of the sweep plane with the edges currently in the slice, and store the order in which the edges were encountered in L.

  Based on the ordering of the contours within L, classify the contours as either solid or hole representations.

  if all contours have not classified
  {
    get the next event from Q
  }
  else all the contours are classified
  {
    break from the classification loop
  }
}

for each labeled and classified contour
{
  determine the output order of the edges in the contours (CCW for solids, CW for holes).
}

output the contours to the laser controller for model fabrication

4.2.2 SLS Process Planner Algorithm Implementation

First, the Y coordinate values of every vertex are stored in descending order, with duplicate entries not being kept. All edges in the slice are then sorted by ascending X coordinate value, where the least value of the X coordinates of the two points in the edge decide where the edge will go in the sorted list. At this point, every edge must be classified as belonging to a contour. By definition, a solid object must be completely closed, so by traversing the edges in a contour you will always return to the initial vertex. If this does not hold for a slice, it is invalid and the process planner cannot be used.
The method of classifying vertices into contours is as follows. Choose an arbitrary edge consisting of points $A$ and $B$. Store $A$ as the initial vertex, and then traverse the edge list to the next edge that contains $B$, and this becomes the next leg in the contour. Repeat the above process until the leg chosen contains vertex $A$ as one of its members. At that time, the contour has been closed and each edge has been classified with a contour number.

Now return to the original edge list, and choose another arbitrary edge that has not yet been classified, and repeat the above process, classifying these edges with a new contour number, and so on until all edges from the slice have been classified into a contour.

Since all edges are now classified into contours, each contour must be classified as a solid or hole. This is done using a sweep line similar to that used for the LADRP planner, and a data structure called a "hit stack".

The sweep line starts at the first (highest) $Y$ coordinate value that was stored above. At this value, each edge in the slice is checked for intersection with the line. If an edge is intersected, its contour number is placed on the end of the hit stack. When a contour is placed on the stack, and it equals the previous entry, that set of two contours is popped off the stack with a solid or hole label.

Since the edges of the slice were previously sorted by ascending $X$ coordinate value, the first edge that is encountered is an outside edge, which means that its contour represents a solid. Given this information, and the fact that a solid contour cannot be inside another solid contour without space between them, determining whether a contour is a solid or a hole can be determined by inspecting the previous contour on the stack. The rules governing the stack and solid/hole assignment are as follows.

Given a set of disjoint contours $A$, $B$, and $C$, three possible stack combinations can occur. All possible combinations can be broken down into these three cases, so these are the only required rules for this portion of the algorithm.

Case 1: \[ A \, B \, B \, A \]
Figure 15 : SLS hit case one

Resolution: Since $A$ is the first entry on the stack, it represents a solid object. Given that $A$ is solid, the next contour encountered must be a hole. When the second $B$ entry is placed on the stack, both $B$ entries are popped and contour $B$ is labeled as a hole.

Case 2: $A A B B$
Resolution: Since A is the first entry, it is labeled as a solid. When the second A entry comes in, they are popped off the stack. When B is then encountered, it becomes the first entry on the stack, and so it also represents a solid, and is handled the same as contour A.

Case 3: \( A B C C B A \)
Resolution: Since $A$ is the first entry, it is labeled as a solid. $B$ is then encountered, and since $A$ is a solid, $B$ must be a hole. Then contour $C$ is hit, and since $B$ is labeled as a hole, $C$ is labeled as a solid. $C$ is then hit again and the two $C$ entries are popped off the stack. The subsequent $B$ and $A$ entries are then governed by cases 1 and 2.

At this point, all vertices have been placed into contours, and all contours have been classified as solids or holes. All that remains is to print the planner output to a text file, the output being the contours in either CCW or CW order.

To determine what order in which to traverse a contour’s edge list to get a CCW or CW ordering is based on the fact that each vertex is connected to exactly two edges in the contour. Connecting to fewer or more edges would invalidate the contour, so this is a safe assumption. Given this information, the lowest vertex (least Y coordinate value) with the least X coordinate value is found. Given this vertex, $A$, it is known that the angle from this vertex to the other vertices in its two edges, $B$ and $C$, is in the range $0 <= \theta < 180$. To determine the traversal order, a vector is created from the $A$ to $B$ and from $A$ to $C$, and the atan of these vectors is computed. The vector that produces the lowest atan value is the edge to start with to get a CCW traversal. Conversely, the vector with the largest atan result is used to produce a CW traversal. Figure 18 below presents a simple pictorial proof of this algorithm.
Figure 18: CW/CCW pictorial proof

All object analysis has now been completed, and the contours may be printed to the text file and sent to the SLS laser controller.
Sort the edges of the slice into ascending X order

while (all edges have not been labeled)
{
    get the next edge in the model
    if the current edge has not been labeled
    {
        Traverse all edges in the contour with the current edge and label with a unique identifier and store them in W.
    }
}

for each contour in the slice
{
    Calculate the intersection of the sweep plane with the edges in W, and store the X values of the intersection points
    sort the list of intersections into ascending order
    classify all the contours in the list as either solid or hole representations based on their ordering within the list
    If all contours have been classified
        exit this loop
}

for every contour in the slice
    determine the CCW/CW ordering of the edges based on the classification of the contours they are a member of.

output the contours to the laser controller for fabrication

4.2.3 SLS Specification Analysis

The algorithm breaks down into three significant parts: generating the event schedule, classifying edges into contours, and labeling each of the contours.
The event queue can be built as a balanced binary tree, (e.g., 2-3 tree or an AVL tree), an operation with a well known complexity of $O(N \log N)$, where $N \leq |V_M|$ is the number of events\cite{Naps86}. Therefore the construction of the event queue takes $O( |V_M| \log |V_M| )$ time.

The contour classification requires a traversal of each edge within $M$. With $|V_M|$ vertices, choose the first vertex and traverse the list of all edges connected to it. Label each edge and vertex touched as part of the same contour, and store them in $W$. Transition to the next unclassified vertex, and repeat the above procedure, until all vertices and edges are labeled as unique contours. Exactly $O(|E_M|)$ edge traversals are needed, giving a complexity of $O(|E_M|)$.

The final portion, contour classification, consists of intersecting the sweep plane with each contour in $M$. At each event point, the sweep plane is intersected with the edges in $M$, and the intersection location is stored. Based on the order of intersection, the contours are labeled. As a worst case, $\log(|V_M|)$ events must be processed, with at most $|E_M|$ edges being intersected at each event. This returns a complexity of $O(|E_M| \log |V_M|)$ for this portion.

This yields a total algorithm complexity of $O(|V_M| \log |V_M| + |E_M| + |E_M| \log |V_M|)$. This equation will be dominated by the $|E_M| \log |V_M|$, since the number of edges in a well defined solid will be greater than the number of vertices.

4.2.4 SLS Implementation Analysis

The algorithm breaks down into four significant parts: generating the event schedule, sorting the edges, labeling the edges as contours, and classifying each of the contours.

The initial task is to obtain all unique vertex $Y$ values and store them in descending order. This is done with an insertion sort, resulting in a complexity of $O(V_M^2)$. All edges in $M$ are then sorted by ascending $X$ coordinate value, again using insertion sort, giving a complexity of $O(E_M^2)$. The contour labeling is identical to that given above, returning a complexity of $O(|E_M|)$.

The final portion, contour classification, consists of intersecting the sweep plane with each contour in $M$. At most, $|V_M|$ events are in the queue, and at most $|E_M|$ traversals could be required per event. This yields an absolute worst case complexity of $O(|V_M| |E_M|)$. The actual operational complexity of this portion will be much smaller, simply due to the fact that most contours will be classified early in the event schedule, but the worst case is possible.

From the above estimates, the overall complexity for the SLS Process Planner Implementation becomes $O( |V_M|^2 + |E_M|^2 + |E_M| + |V_M| |E_M| )$. 

51
5 Results

The tools that were developed as a part of this research have proven to be an excellent start towards the goal of easy and efficient desktop manufacturing. With a minimal amount of time, a user is able to design a BREP model capable of representing a complex solid object in Unigrafix. Properly using the Slicer application and the LADRP setup at CREOL, the largest time investment in the entire production process comes from the actual laser construction of the prototypes.

As research into the physical properties of the laser and material systems progress, this time investment will greatly decrease, while the quality of the prototypes will gradually approach actual production level.

The current Slicer is capable of handling all well constructed BREP polygonal objects whose winged edge structure represents a planar graph. The LADRP system at CREOL is currently only able to handle a subset of these objects due to difficulties with the material feeder process. However, research currently being conducted by CREOL and other similar institutions continue to provide hope for a seamless and efficient system capable of producing high quality objects. Figures 19 - 24 below illustrate some sample objects that have been handled by the slicer and laser process controllers.
Figure 19: Sample model #1
Figure 20: Slices of model #1
Figure 21: Sample model #2
Figure 22: Slices of model #2
Figure 23: Sample model #3
Figure 24: Slices of model #3
6 Conclusion

The multilevel design paradigm followed in VLSI design is possible only with a precise and clean interface between the layers of the design. The observations gathered over the years from experiences in VLSI research seem to strongly lend themselves to the family of manufacturing processes known as SFF and MEMS. This research was undertaken specifically with the goal of providing a suite of tools that would lead to the development of a possible structured design methodology for SFF processes.

Of specific interest to this thesis are the manufacturing and laser-aided processes of LADRP and SLS. The advent of these new layering based technologies have opened entirely new areas of possibility for rapid prototyping and distributed manufacturing. A link has also been created from these new technologies to the time tested field of VLSI through the use of the basic design hierarchies that were at the very core of the VLSI revolution.

With this blend of old and new concepts, several areas of interest have been opened. The software tools and algorithms in this thesis hope to address a few of these areas, specifically through the development of the Slicer, and tool capable of layering a 3D object, and the LADRP and SLS Process Planners, tools able to accept a BREP layer model and translate it into a unique series of commands understandable by the laser processors.

The initial results that have been obtained from this research seem to strongly indicate that the layering paradigm of VLSI can be effectively applied to a manufacturing process. With time and research, the involved processes will continue to improve until such a time as the creation of a prototype will require less time that it took to think up the idea for the prototype in the first place.
7 Future Work

As with any body of work, for every item implemented, numerous new ideas pop up. The sections below describe a few areas in which this research could be continued.

7.1 Unigrafix GUI

As mentioned above, Unigrafix files are designed in text files. At times, this can be a cumbersome way to design an object. The file must be created, then viewed with UG, then back to the text file for modifications, and so on until the model is properly designed. It would be more efficient if the user were able to directly create and modify a model from a GUI, able to see the results during the design phase.

Creating a 3D GUI presents a number of problems. Issues such as viewing perspectives during design, how much input must be text, and how much directly to the GUI, how modifications will be done, and what internal representation will be used for the models. There are many commercial CAD packages that deal with these issues in a variety of ways. It would be possible to simply use a CAD package as a front end and have an interface between the GUI and the slicer that converts the model into UG format. This approach may be easier than a completely new GUI, but that is an issue for some other research.
7.2 BREP to CSG Converter

At this time, Unigrafix and the slicer operate exclusively on BREP models. It would be useful to have a converter capable of taking slicer output and translating it into CSG representations.

Other fabrication processes than the ones in this research operate on CSG models instead of BREP. A converter would allow the slicer to operate as is, and then send its output through the converter, thus extending its abilities to include a new family of fabrication processes.

7.3 Curved Surfaces

At this time, Unigrafix does not support the representation of curves, thus limiting the slicer to polygonal objects. This is a serious limit to impose on the design and fabrication processes. If a curved surface is desired in a model, the designer must create the model using polygonal approximates of the curve, thus limiting the validity of the model. As a minimum, support would be needed for splines and NURBs in order to accurately model curved surfaces.

7.4 Adaptive Slicing

It is of the utmost importance that when an object is sliced, the amount of information lost is minimized. The prime example of this information loss is when taking a slice of an object that has a high degree of curvature along the slice axis. Since slicing produces 2D linear cuts, the curve of the object is effectively lost, being replaced with an approximation as the slices are layered. This effect is called stair stepping, and can lead to incomplete or incorrect prototype production [Requicha85].

A second problem deals with the areas of significance within an object. An area of significance is best defined as any section of a model that contains non-homogeneous height information within the bounds of a slice. All layering processes build objects by depositing subsequent layers on top of previous ones, essentially building from the ground up. Since this is accomplished by using 2D slices, a single layer must be of uniform height, or model integrity could be lost. To surmount this problem, the slices of an object must contain only a single 2D layer of significance.

A final issue is that of the orientation of the object prior to slicing. All slices must be made along the same axis, but all axis choices will not produce the same quality of results. By allowing the ability to choose the axis along which the slice proceeds, the problems given above can be alleviated somewhat, or possibly even eliminated.

Unfortunately, allowing the slicer to determine the best slice size and object orientation is a complex task and outside the scope of this thesis. To solve this problem, the user is given the responsibility/ability to choose the proper slice axis and the size of the slices to be processed. In this way the model information lost between design and fabrication is kept to a minimum.
Bibliography


62


